Towards Semi-automated Parallelization of Data Stream Processing

Martin Kruliš, David Bednárek, Zbyněk Falt, Jakub Yaghob and Filip Zavoral

Abstract Current hardware development trends exhibit clear inclination towards parallelism. Multicore CPUs as well as many-core architectures such as GPUs or Xeon Phi devices are widely present in both high-end servers and common desktop PCs. In order to utilize the computational power of these parallel platforms, the applications must be designed in a way that intensively exploits parallel processing. In our work, we propose techniques that simplify the application decomposition process in data streaming systems. The data streaming paradigm may be applied in many data-intensive applications, e.g., database management systems or scientific data processing. In order to employ these techniques, we have developed a data streaming language called Bobolang that simplifies the design of the application. This approach allows the programmer to write strictly serial operators in a traditional language and then interconnect these operators in an execution plan, that presents opportunities for automated parallel processing.

1 Introduction

Streaming systems represent a specific domain of computing environment. These systems operate with data streams, which are basically unidirectional flows of structured tuples. Streams are processed by operators (also denoted functions, kernels, or

M. Kruliš (✉) · D. Bednárek · Z. Falt · J. Yaghob · F. Zavoral
Charles University in Prague, Prague, Czech Republic
e-mail: krulis@ksi.mff.cuni.cz

D. Bednárek
e-mail: bednarek@ksi.mff.cuni.cz

Z. Falt
e-mail: falt@ksi.mff.cuni.cz

J. Yaghob
e-mail: yaghob@ksi.mff.cuni.cz

F. Zavoral
e-mail: zavoral@ksi.mff.cuni.cz

P. Novais et al. (eds.), *Intelligent Distributed Computing IX*,
Studies in Computational Intelligence 616,
DOI 10.1007/978-3-319-25017-5_22

filters) which may have multiple inputs and outputs. These operators transform data from the input streams by performing their built-in functionalities and pass their results into the output streams. The operators are usually implemented in a procedural or object-oriented programming language such as C/C++ and compiled natively.

A streaming application is typically represented as an oriented graph, where the vertices are operators and the edges prescribe the data flow between them. In the remainder of this paper, we will refer to this graph as the *execution plan*. The execution plan is usually described in specialized declarative languages.

There is a large number of existing streaming systems [2–4, 10, 11, 14, 16], while each is designed for a specific purpose or for a different platform. The streaming systems were originally designed for scenarios, where the data naturally occur as a stream (e.g., sensory data) and where continuous processing is required. However, the streaming paradigm can be also used to express parallelism in a more programmer-friendly way. In this context, we recognize two types of parallelism:

- the *inter-operator parallelism* (concurrent processing of multiple operators), and
- the *intra-operator parallelism* (parallelism within one operator).

The *inter-operator parallelism* emerges quite naturally in the streaming systems. It only requires that the operators are truly independent and that there are enough data fragments to keep multiple operators occupied. The *intra-operator parallelism* cannot be achieved automatically by the streaming system task scheduler, since the operators are usually treated as indivisible blocks of code. However, in some cases, we can decompose an operator into multiple sub-operators which perform the same functionality. The decomposed structure leads to a larger execution plan; hence, it presents more opportunities for inter-operator parallelism.

We narrow our focus on shared memory streaming systems which implement the data streams as flows of packets and explicitly expose this implementation to the programmer. Typical representatives of such systems are *The Flow Graph* component from the Intel Threading Building Blocks [13] or the *Bobox* framework [2]. The packet-level processing in shared memory permits certain optimizations, especially when the packets are passed on without modification, or when the data are shallow copied.

In this paper, we address the issues of effective semiautomated parallelization in the streaming systems. These systems separate the design of effective code (reduced to simple serial routines) from the data flow schema where the potential parallelism is expressed along with requirements for implicit synchronization. Furthermore, we introduce techniques how to decompose operators in order to express intra-operator parallelism (within one operator) by the means of inter-operator parallelism (concurrent processing of operators).

The proposed techniques were implemented in Bobolang language [7], which was designed for the specification of execution plans. This language is integrated into Bobox framework [2], which provides a runtime for parallel evaluation of the plans on shared memory systems. This framework is primarily designed for efficient parallel data processing, thus it provides an excellent platform for our experiments.

The paper is organized as follows. Related work is collected in Sect. 2. Section 3 presents our approach towards semi-automated parallelization in streaming systems. The Bobolang language which implements this concept is described in Sects. 4 and 5 concludes the paper.

2 Related Work

Contemporary streaming languages basically differ in their focus designed with a particular intent which significantly influences their syntax and semantics. The languages Brook [3], StreaMIT [14] and StreamC [6] are intended for the development of efficient streaming applications. They introduce a language based on the C/C++ syntax, which allows the programmer to implement the operators and to specify their mutual interconnections. The compiler exploits the streaming nature of the application to perform specific analyses and optimizations designed with a particular emphasis on concurrent execution. The compiler also creates a static mapping of the operators to the execution units such as CPUs, GPUs, or FPGAs.[1]

Another language extension designed for the development of streaming applications is Granular Lucid (GLU) [9]. The operators are implemented in the C language; their structure is described in Lucid. The parallel evaluation of operators is designed in a similar way as in Bobolang.

The X Language [8] is another example of modern streaming language. It is logically similar to GLU, but it uses different syntax (similar to Bobolang) which clearly and explicitly describes the connections between operators. This is especially useful for designing complex algorithms.

The main difference between these languages and our approach is that they lack support for constructions such as multiplication of inputs/outputs (see Sect. 4.1). The absence of this feature requires the use of parallelism inside the procedural code of operators; otherwise, only inter-operator parallelism would be available.

Semiautomatic parallelization is usually studied in a context of particular programming languages such as C (Paralax [15]) or Python (Pydron [12]) where the sequentially programmed source code is transformed into parallelizable pieces of code. FastFlow [1] accelerator supports the easy porting of existing sequential C/C++ applications onto multi-core systems. Code kernels identified by a programmer are offloaded onto a number of additional threads running on the same CPU.

Despite the abundance of parallelism in streaming applications, it is a nontrivial task to split and efficiently map sequential applications to multicore systems. Cordes et al. presents an algorithm [5] which automatically extracts pipeline parallelism from sequential ANSI-C applications. This method employs an integer linear programming (ILP) based approach to automatically control the granularity of the parallelization.

[1] Field-programmable gate array.

3 Semi-automated Parallelization

Streaming systems naturally introduce concurrent processing of operators, which is called inter-operator parallelism. The inter-operator parallelism is usually cooperative (i.e. non-preemptive), associated to sending and receiving data between operators, where an incoming packet triggers the execution of the operator (if it was suspended after finishing the previous work). To balance the load among available processors, the operator code is usually allowed to migrate across a pool of worker threads. In other words, the incoming data packets generate a sequence of tasks which are assigned to worker threads by the underlying scheduler. This mode of concurrency is often called *pipeline parallelism* and it is sometimes considered a special case of *task parallelism*.

Although some systems allow parallel code inside individual operators, parallel programming is difficult for the developer. Instead of implementing intra-operator parallelism by parallel code inside an operator, we suggest multiplication of the operators in the execution plan and inserting auxiliary operators to dispatch the input data among the replicas of an operator and to collect the output data. This way, the available parallelism is explicitly denoted in the execution plan and the individual copies of the operator may remain sequential.

Of course, the multiplication of an operator must be consistent with its behavior and the way the data are dispatched. The simplest case, described in Sect. 3.1, is associated with *stateless* operators which process each packet of data independently. If the operator depends on its internal state, a copy of the state must be properly maintained in every replica of the operator. As we will show in Sect. 3.2, there are situations where the cost of maintaining the replicated state is significantly lower than the gain of the parallelization.

The cost of dispatching data and maintaining replicated state depends on the cost of data transfer between the operators. Consequently, our approach aims at systems which use shared memory for communication between operators and allow cheap broadcasting and forwarding of packets via sharing memory regions. In addition, we rely on the ability of the underlying scheduler to balance the load among processors. Thanks to this ability, the auxiliary operators are not required to balance the load exactly since minor skewness will be corrected by the scheduler.

In our approach, the designer of the execution plan decides which operators may be parallelized and marks them as *stateless* or *parallelizable*. In the latter case, the procedural code of the operator must adhere to the protocol described in Sect. 3.2. The execution plan will then be automatically transformed by multiplication of selected operators and insertion of auxiliary operators.

Besides the two built-in approaches to parallelization, the plan designer may explicitly invoke the multiplication of operators, using plan annotations described in Sect. 4.

3.1 Data Parallelism

Data-parallel subproblems would be implemented by a *parallel for* in traditional parallel code. In pipeline systems, such subproblems correspond to stateless operators. Thanks to the absence of internal state, we may split the input stream into several sub-streams and process each sub-stream independently, by identical replicas of the original operator. Two auxiliary operators are required, as shown in Fig. 1: The *dispatch operator* distributes incoming packets to its outputs in round robin manner and the *consolidate operator* interleaves incoming packets. The auxiliary operators have negligible overhead, since they only forward incomming packets.

3.2 Maintaining the Local State

If an operator maintains local state, the parallelization process requires minor modification of the internal function of the operator as well as different data dispatching scheme. The parallelization comes at the cost of performing redundant work. It depends on the nature of the operator whether the cost is acceptable, i.e. lower than the gain by parallelism. The plan designer has to assess the overhead and decide whether an operator should be parallelized this way. Let us consider a typical schema of a general stateful operator with an internal state S:

$S \leftarrow$ initial state
while not finished **do**
    tuples $\leftarrow$ next part of input (e.g., next packet)
    process *tuples* whilst using and updating state S
end while

If the processing of the tuples and the update of the state can be effectively separated from each other and the updating of state S takes significantly less time than the processing of tuples, the stateful operator can be effectively parallelized. The concurrency is achieved by replicating the operator while each replica has its own copy of the state. Each replica has a unique index from 0 to $R - 1$ (for R replicas) called *RID* (Replica ID). Each of the replicas then perform the following algorithm:

$S \leftarrow$ initial state, $\quad phase \leftarrow 0$
while not finished **do**

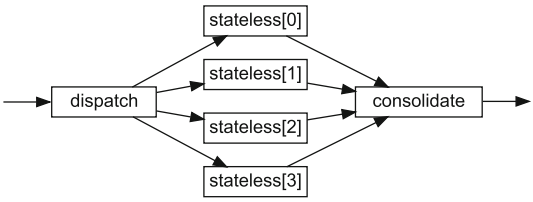

Fig. 1 Parallelization of a stateless operator

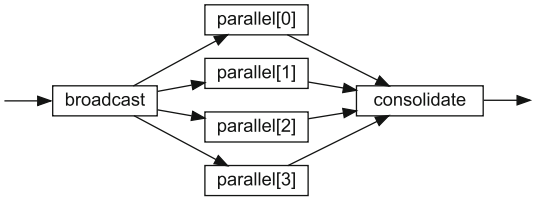

Fig. 2 Parallelization of a stateful operator (RIDs are in brackets)

tuples ← next part of input (e.g., next packet)
if *phase* mod $N = RID$ **then**
process *tuples* whilst using and updating state *S*
else
update state *S* using *tuples*
end if
phase ← *phase* + 1
end while

We denote operators which are modified in this way as *parallelizable*.

The schema is depicted in Fig. 2. The *broadcast operator* which clones its input for all its outputs is used here instead of the dispatch operator used in the stateless case. The data are efficient shallow copied in the shared memory. All the operator replicas receive identical (shallow) copies of the original stream, but they alternate in the processing of the tuples and in the production of the output stream. The resulting stream is then gathered by the *consolidate* operator as in the case of parallelization of the stateless operators. The body of the operator must be designed by the developer to support this parallelization. On the other hand, many problems allow simple decoupling of the tuple processing and state updates simply by creating conditions in existing code.

4 The Bobolang Language

Bobolang is a declarative language designed for specification of execution plans—together with the procedural code of individual operators, the execution plans forms a parallel application. The application requires a runtime environment, essentially consisting of a dynamic scheduler and streaming support. In our case, the runtime environment is Bobox [2] where the primary procedural language for operators is C++; however, Bobolang itself is independent of both the runtime and the associated procedural language and the same plan may even be used with different implementations of the operators. For type safety, the Bobolang compiler is supplied with a dictionary of *column types* which are provided by the runtime.

The features of Bobolang are shaped by the following objectives:

- Providing a hierarchical decomposition of the execution plan. The operators referenced in the *main* plan may either be *atomic* operators implemented in the associated procedural language or *compound* operators whose interior is defined by a *model* defined in a model library. All the models are again defined in Bobolang, allowing for unlimited (but not recursive) decomposition into a tree-like hierarchy of models with atomic operators at leaves.
- Allowing generic models independent of concrete column types. Similarly to functions in generic programming, generic models infer the number and types of columns at their interfaces from the context of their instantiation.
- Multiplication of inputs, outputs, and operators as a means to provide opportunity for parallelization. The degree of multiplication is either specified directly in the model or set to defaults provided from the outer environment. The defaults are set based on the parallel processing hierarchy of the hardware, considering available threads, cores, CPU sockets, caches, and/or NUMA nodes.
- Automatic multiplication of stateless and parallelizable operators, using insertion of built-in operators *broadcast*, *dispatch*, or *consolidate*.

A definition of a compound operator is illustrated in the following example:

```
operator new_operator(int)->(int,int) {
  split_op(int)->(int),(int) split;
  filter_op(int)->(int,int) filter1, filter2;
  join_op(int,int),(int,int)->(int,int) join;

  input -> split;
  split[0] -> filter1 -> [0]join;
  split[1] -> filter2 -> [1]join;
  join -> output;
}
```

The first line declares a new operator called `new_operator`. The operator has one input (a stream of integers) and one output (a stream of integer pairs). The body of the operator has two parts. The first part contains a list of sub-operators, i.e. instances of nested operators from which the operator is composed of. Each line specifies the operator type together with its input/output data descriptor and declares one or more local identifiers of the nested operator instances.

The second part specifies the connections between operators. Statement `op1 -> op2` defines the connection of `op1` output to `op2` input. The corresponding input and output must have the same data type descriptor. The syntax allows creating chains, so the `op1 -> op2 -> op3` statement is just a shorthand expression for `op1 -> op2` and `op2 -> op3` statements.

In addition to explicitly defined sub-operators, each body implicitly contains two special sub-operators—`input` and `output`. These sub-operators represent the input and the output of the operator `new_operator`.

Operators may have multiple inputs or outputs. The inputs/outputs are indexed by consecutive numbers starting with zero. The index of an output is written in brackets as a suffix of the identifier of the operator, the index of an input is written analogically

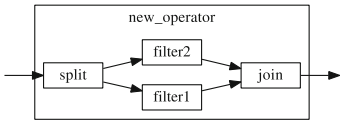

Fig. 3 Internal structure of the `new_operator`

as a prefix. If an operator has only one input/output, the index may be omitted. Note that the `(int),(int)` denotes two streams of integers, whereas `(int,int)` denotes one stream of integer pairs. The resulting internal structure is depicted in Fig. 3.

4.1 Multiplication of Inputs and Outputs

Some operators have the ability to split (or broadcast) their output into N channels while other operators can receive their input from multiple channels. Splitting creates the opportunity for parallelism as the operators between the splitting and merging operators are multiplicated and run in parallel. Bobolang allows the specification of multiplicated outputs and inputs and handles the replication of the operators. On the other hand, the method of splitting (round-robin, hash-based, etc.) as well as merging (ordered, random) is a matter of agreement between the operators involved and must be consistent with the properties of the operators in between.

The main objective of the multiplication mechanism is to allow plan description which does not grow with the degree of multiplication and where the degree of multiplication may be either be specified explicitly or inferred from the environment. The mechanism also allows the propagation of multiplication from the main plan to the lower plans in the model hierarchy, allowing internally parallelized sub-models to communicate via multiplied channels.

In a Bobolang plan, each input and output may be augmented with a *multiplier*, either *explicit* (a number in curly brackets) or *implicit* (an asterisk). In both cases, the multiplier signalizes the ability of the operator to produce or consume multiplicated channels, with the specified or an arbitrary degree. As a result, both channels and operators are multiplicated and the Bobolang compiler tries to find an equilibrium which satisfies the following equation for each edge connecting operator the ith output of the operator op_a to the jth input of the operator op_b:

$$d(op_a) \cdot d(out_{a,i}) = d(in_{b,j}) \cdot d(op_b)$$

Here, $d(op_a)$ and $d(op_b)$ are the degrees of multiplication of the two operators while $d(out_{a,i})$ and $d(in_{b,j})$ are the degrees of output/input multiplication at the edge ends. The products at both sides of the equation correspond to the degree of multiplication of the connecting channel. A part of the output/input degrees may be set

by explicit multipliers in the source plan, others are set to 1 where no multiplier is specified. The degrees associated with implicit multipliers (asterisks) as well as the degrees of operators are computed by the Bobolang compiler. Whenever the equations allow a degree of freedom, implicit multipliers are set to default values from the environment. Once all the input/output degrees are fixed, the degrees of operators become uniquely determined by propagation from the main plan borders whose multiplicity degrees are set to 1. Care must be taken when explicit multipliers are used, because the system of equations may become overconstrained and thus no solution may be found.

4.2 Intra-operator Parallelization

We described two types of operators in Sect. 3 and methods of parallelizing them. To make the parallelization process easier, we introduce Bobolang keywords that specify the type of a sub-operator, so the Bobolang interpret can select appropriate parallelization method.

The `stateless` keyword informs the compiler that the sub-operator instance does not have inner state, so it can be always parallelized. The `parallel` keyword denotes operators that are stateful, but have been modified in the way prescribed by our schema and the programmer explicitly requests that they are parallelized.

```
stateless stateless_op()->() op1;
parallel parallelizable_op()->() op2;
```

If we mark the operator as `stateless`, the operator is automatically replaced with the following schema (with respect to the number of inputs/outputs):

```
operator parallelized_stateless (in_type)->(out_type) {
    dispatch(in_type)->(in_type)* disp;
    stateless_op(in_type)->(out_type) op;
    consolidate(out_type)*->(out_type) cons;

    input -> disp -> op -> cons -> output;
}
```

Therefore, when the operator `stateless_op` is used in an execution plan, it is decomposed as shown in Fig. 1. The `parallel` keyword uses very similar schema. The only difference is that it employs a `broadcast` operator instead of `dispatch` operator as depicted in Fig. 2.

5 Conclusion

In this paper, we have presented an elegant concept of semi-automated parallelization designed for data streaming systems. This concept streamlines the implementation of the core functionality of data processing systems whilst providing a safe way how to introduce various forms of parallelism into an application.

Data parallel subproblems with no internal state may be parallelized directly by the means of the replication scheme for stateless operators. The concept of stateless operator is simpler to handle and less error prone, since the programmer designs the internal functionality regardless of the level of parallelism. For more complex cases, we have proposed the concept of parallelizable stateful operator which permits parallelization at the cost of redundant work.

The presented concepts has been implemented in the Bobolang language and successfully applied in the implementation of parallel database engines. The underlying Bobox system is currently being extended to parallel accelerators such as GPUs and Xeon Phi devices. Their unique properties become an impulse for further development of the Bobolang language and the presented parallelization concepts.

Acknowledgments This work was supported by the Czech Science Foundation (GACR) projects P103-14-14292P and P103-13-08195S and by Specific Research SVV-2015-260222.

References

1. Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M.: Accelerating code on multi-cores with fastflow. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011 Parallel Processing. Lecture Notes in Computer Science, vol. 6853, pp. 170–181. Springer, Berlin (2011)
2. Bednarek, D., Dokulil, J., Yaghob, J., Zavoral, F.: Bobox: parallelization framework for data processing. In: Advances in Information Technology and Applied Computing (2012)
3. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for GPUs: stream computing on graphics hardware. ACM Trans. Graph. **23**, 777–786 (2004)
4. Consel, C., Hamdi, H., Réveillère, L., Singaravelu, L., Yu, H., Pu, C.: Spidle: a DSL approach to specifying streaming applications. In: Proceedings of the 2nd International Conference on Generative Programming and Component Engineering, pp. 1–17. Springer, New York, NY, USA (2003)
5. Cordes, D., Heinig, A., Marwedel, P., Mallik, A,: Automatic extraction of pipeline parallelism for embedded software using linear programming. In: Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on, pp. 699–706 (2011)
6. Das, A., Dally, W.J., Mattson, P., Compiling for stream processing. In: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques, pp. 33–42. ACM, New York, NY, USA (2006)
7. Falt, Z., Bednárek, D., Kruliš, M., Yaghob, J., Zavoral, F.: Bobolang: a language for parallel streaming applications. In: Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing, pp. 311–314. ACM (2014)
8. Franklin, M., Tyson, E., Buckley, J., Crowley, P., Maschmeyer, J.: Auto-pipe and the X language: a pipeline design tool and description language. In: 20th International Parallel and Distributed Processing Symposium. IEEE (2006)

9. Jagannathan, R., Dodd, C., Agi, I.: Glu: a high-level system for granular data-parallel programming. Concurrency—Pract. Expe. **9**(1), 63–83 (1997)
10. Kapasi, U.J., Dally, W.J., Rixner, S., Owens, J.D., Khailany, B.: Programmable stream processors. IEEE Comput. **36**, 282–288 (2003)
11. Mark, W.R., Steven, R., Kurt, G., Mark, A., Kilgard, J.: Cg: a system for programming graphics hardware in a C-like language. ACM Trans. Graph. **22**, 896–907 (2003)
12. Muller, S.C., Alonso, G., Amara, A., Csillaghy, A.: Pydron: semi-automatic parallelization for multi-core and the cloud. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), pp. 645–659. USENIX Association (2014)
13. Reinders, J.: Intel threading building blocks. O'Reilly, Sebastopol (2007)
14. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: a language for streaming applications. In: Compiler Construction, pp 179–196. Springer (2002)
15. Vandierendonck, H., Rul, S., De Bosschere, K.: The paralax infrastructure: automatic parallelization with a helping hand. In: Parallel Architectures and Compilation Techniques, 19th International Conference, Proceedings, pp. 389–400. Association for Computing Machinery (ACM) (2010)
16. Zhang, D., Li, Z.Z., Song, H., Liu, L.: A programming model for an embedded media processing architecture. In: Embedded Computer Systems: Architectures, Modeling, and Simulation, pp. 251–261. Springer (2005)